The Parallel Research Kernels, a tool for parallel systems investigations - Part I

Rob Van der Wijngaart

Intel Labs

https://github.com/ParRes/Kernels

AMS Seminar Series, NASA Ames Research Center, October 13, 2016

Parallel system=hardware system+network stack+OS+parallel programming environment***
(ProgEnv: programming model + API + compiler + runtime)

Motivation

Observations

- Performance of full app mixture of multiple effects/interactions: hard to apply learnings to other apps
- Hard to obtain useful data of full app on simulator (1 s * 1M = 11.6 days)
- Can't predict which apps (or languages, or ProgEnvs) important in 10 years
- **But:** Can predict which fundamental parallel constructs/patterns will matter

Proposal: provide something simpler

- Generic parallel-specific app patterns, i.e. **parallel kernels**
- Each kernel is dominated by only one pattern

Agenda

- Motivation
- Limitations
- Philosophy
- Context
- Usage model
- Reference implementations
- PRKs you should care about
- PRK you may care about
- Example results

Agenda

- Motivation
- Limitations
- Philosophy
- Context
- Usage model
- Reference implementations
- PRKs you should care about
- PRK you may care about
- Example esults

Limitations

- Focused (mostly) on features stressed by **parallel** parts of application, emphasizes parallel overhead, so may **exaggerate** parallelization impact
- Not designed for full application performance projections
- Single data structure, one or two hot loops: small **data layout**/alignment details may dominate performance
- Not designed to measure **robustness**: fault tolerance, I/O performance
- Not designed to measure ProgEnv **productivity**, due to kernel simplicity
- Not designed to measure ProgEnv **expressiveness**; that battle had been fought … we thought

Agenda

- Motivation
- Limitations
- Philosophy
- Context
- Usage model
- Reference implementations
- PRKs you should care about
- PRK you may care about
- Example results

Philosophy

- Broad range of **important patterns** found in real parallel applications
- Reasonably self-contained for **all of HPC**
- Paper-and-pencil **specifications**
- **Simple**, understood by non-domain scientists (not *algorithms,* but *patterns*!)
- Each kernel does some real work (data transformation). Corollaries:
  - Uniform performance metric = work/time
  - Work can be tested for correctness
- Compact reference codes *O*(1-3 pages): easy porting to new ProgEnv
- Performance expectations (simplified performance models)

Agenda

- Motivation
- Limitations
- Philosophy
- Context
- Usage model
- Reference implementations
- PRKs you should care about
- PRK you may care about
- Example results

Context

PRK are nothing new; PRK are different

Legend:
✓: yes
~: meh
—: no
? : dunno

	NPB	CLOMP(I)	EPCC	HPCC	SPEC MPI	SPEC OMP	PRK
arbitrary scale	—	✓	✓	✓	—	—	✓
verification	✓	~	—	✓	✓	✓	✓
many runtimes	—	—	—	~	—	—	✓
pattern coverage	~	—	—	—	?	?	✓
compact	—	✓	✓	~	—	—	✓
work metric	✓	~	—	✓	✓	✓	✓
performance model/ expectation	—	—	—	—	—	—	✓

PRK are like the English language: steal stuff from wherever you can and make it your own

Agenda

- Motivation
- Limitations
- Philosophy
- Context
- Usage model
- Reference implementations
- PRKs you should care about
- PRK you may care about
- Example results

Usage model

- Analyze app, map patterns to kernels, study performance of kernels
- If system **does well** on **all** relevant kernels, move to mini-app or actual application (method of **elimination**)
- Example parallel application analysis:
  1. read lists of data
  2. do local sort into buckets
  3. send one bucket each to all other nodes
  4. merge incoming buckets
- Useful PRKs: 1: Nstream, 2: Sparse, Random, or Refcount, 3: Transpose, 4: Nstream

Agenda

- Motivation
- Limitations
- Philosophy
- Context
- Usage model
- Reference implementations
- PRKs you should care about
- PRK you may care about
- Example esults

Reference implementations

- Portable:
  - plain C/Fortran serial reference implementations, no excessive tuning
  - no assembly/intrinsics/ libraries (except MKL's DGEMM, optional)
- Multiple parallel versions:
  - "Traditional": OpenMP, MPI: one- and two-sided + hybrid (OpenMP, MPI3 SHM), CAF,AMPI, FG-MPI
  - Disruptive: Charm++, Grappa, UPC, OpenSHMEM, Legion, HPX, OCR, Chapel, HClib, …
  - Oddball: Julia, Python
- Parameterized: problem size, #iterations, algorithmic choices
- No input files; all initialization data synthesized
- Automatic verification test: robust, nonintrusive, inexpensive
  - keeps users honest
  - facilitates porting/debugging

CAF=Fortran with co-arrays, OCR=Open Community Runtime, AMPI=Adaptive MPI, FG-MPI=Fine-grain MPI

Agenda

- Motivation
- Limitations
- Philosophy
- Context
- Usage model
- Reference implementations
- PRKs you should care about
- PRK you may care about
- Example results

PRKs you should care about

Because they:

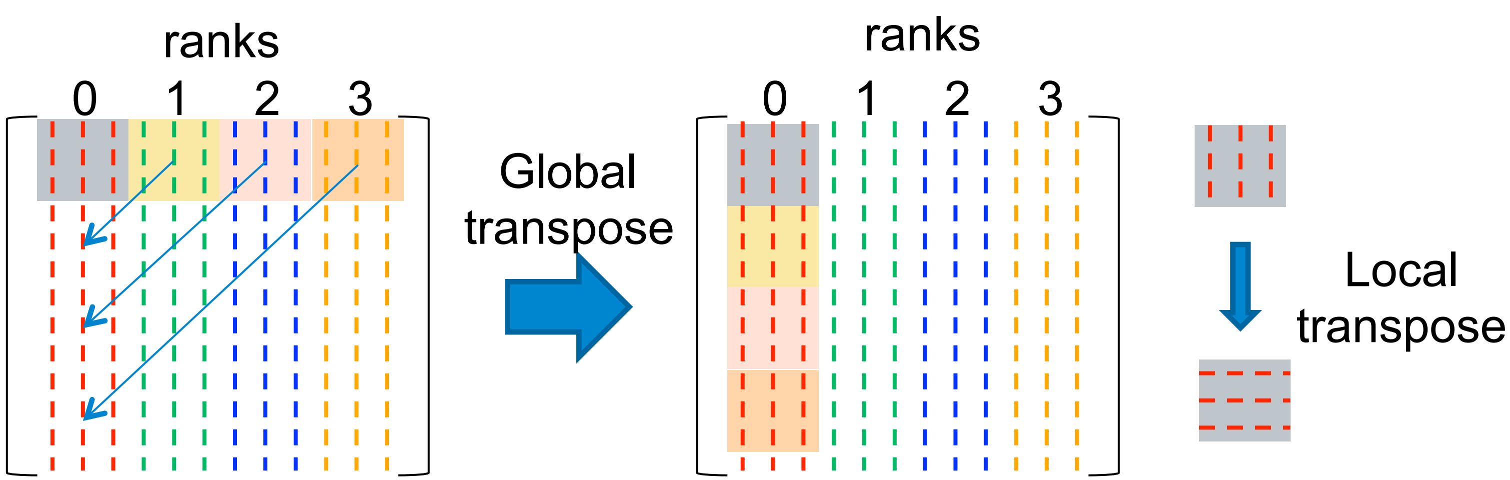
- Exhibit a range of granularities
- Feature drastically different communication patterns
- Proxy very important patterns in HPC
- Contain both data parallel and non-data-parallel patterns

Dense matrix transposition (transpose)

Operation: A += (B++)^T, A and B distributed identically, whole columns, column-major storage format

Proxy for: global data redistribution (cf FFT)

Granularity from very coarse to very fine, especially with strong scaling

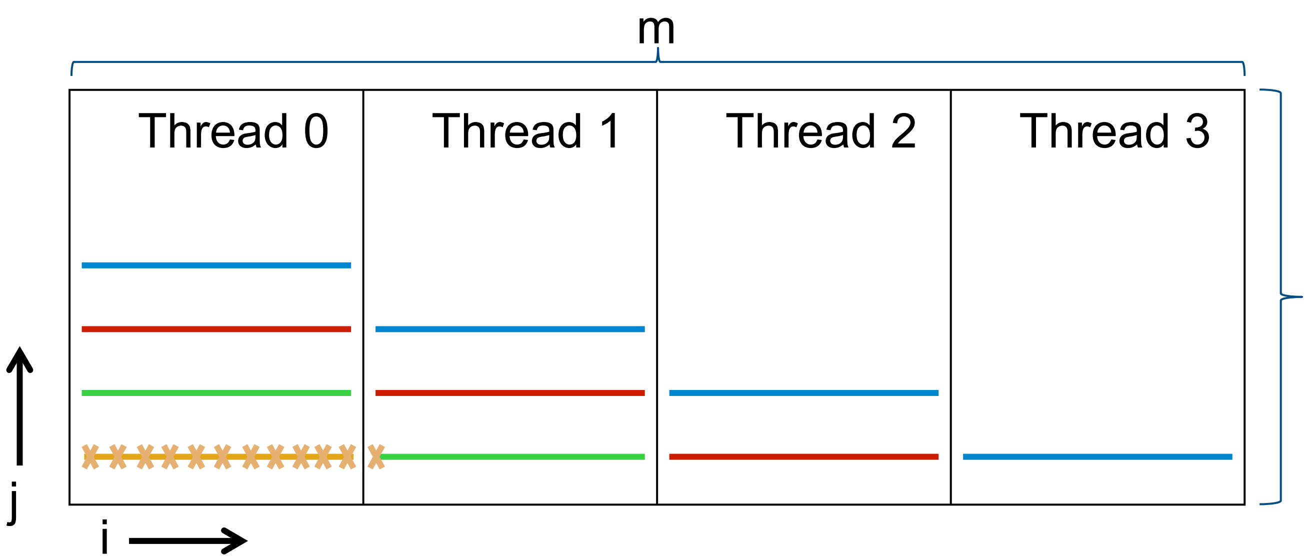


Point-to-point synchronization (*synch_p2p*)

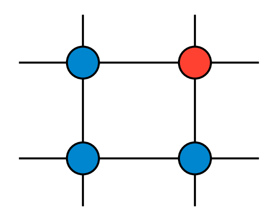
Operation: $A(i,j) = A(i-1,j) + A(i,j-1) - A(i-1,j-1)$

$A(0,0) = -A(m-1,n-1)$ [to couple successive sweeps over the grid]

Proxy for: pipelined solution of problem with non-trivial 2-way dependencies



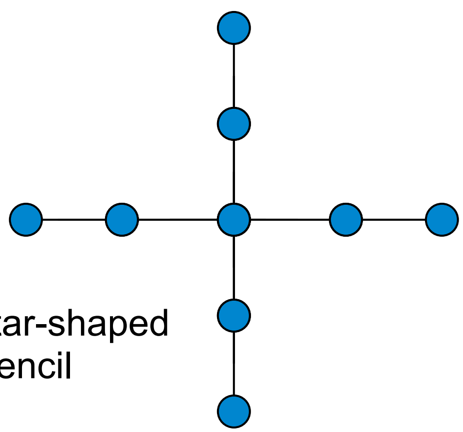
Granularity very fine, no data parallelism



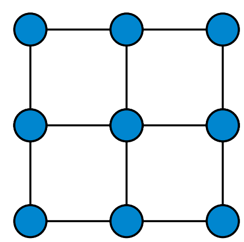
Data parallel stencil (stencil)

Operation: For all points in 2D grid, compute a += S(b++), where S is a stencil operation (box or star-shaped), a and b are scalar grid variables (2D arrays)

Proxy for: multi-dimensional array operations with spatial locality



Star-shaped stencil



Box-shaped stencil

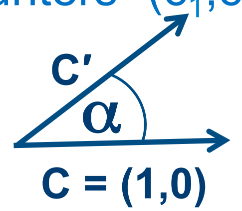
Granularity medium, reuse factor depends on radius of stencil (tunable)

Reference counting (refcount)

Operation: Update pair(s) of reference “counters” (c_1,c_2) in tandem

Independent: (c1 c2)′ = (c1 c2)++

Coupled: (c1 c2)′ = R(α) (c1 c2)



Proxy for: mutual exclusion, high and low contention, simple and compound

- Counters can be integer (independent only) or floating point
- Mutex can be atomic, lock, or none
- Counter updates can be overlapped with independent work (tunable)
- Counters can be privatized (uncontended locks)

Agenda

- Motivation
- Limitations
- Philosophy
- Context
- Usage model
- Reference implementations
- PRKs you should care about
- PRK you may care about
- Example results

PRKs you may care about

Because they:

- Test some additional important synchronization constructs
- Provide information about serial performance and compiler smarts

The list:

- DGEMM (MKL or hand-coded): top flops
- Nstream: top memory bandwidth
- Synch_global: global synchronization (OpenMP barrier/MPI_Allgather)
- Sparse: Sparse matrix-vector multiply: memory latency
- Random: HPCC Random Access, fixed verification + small problem sizes: latency
- Reduce: vector reduction
- Branch: inner loop conditionals (vectorization), PC jumps, instruction cache

Agenda

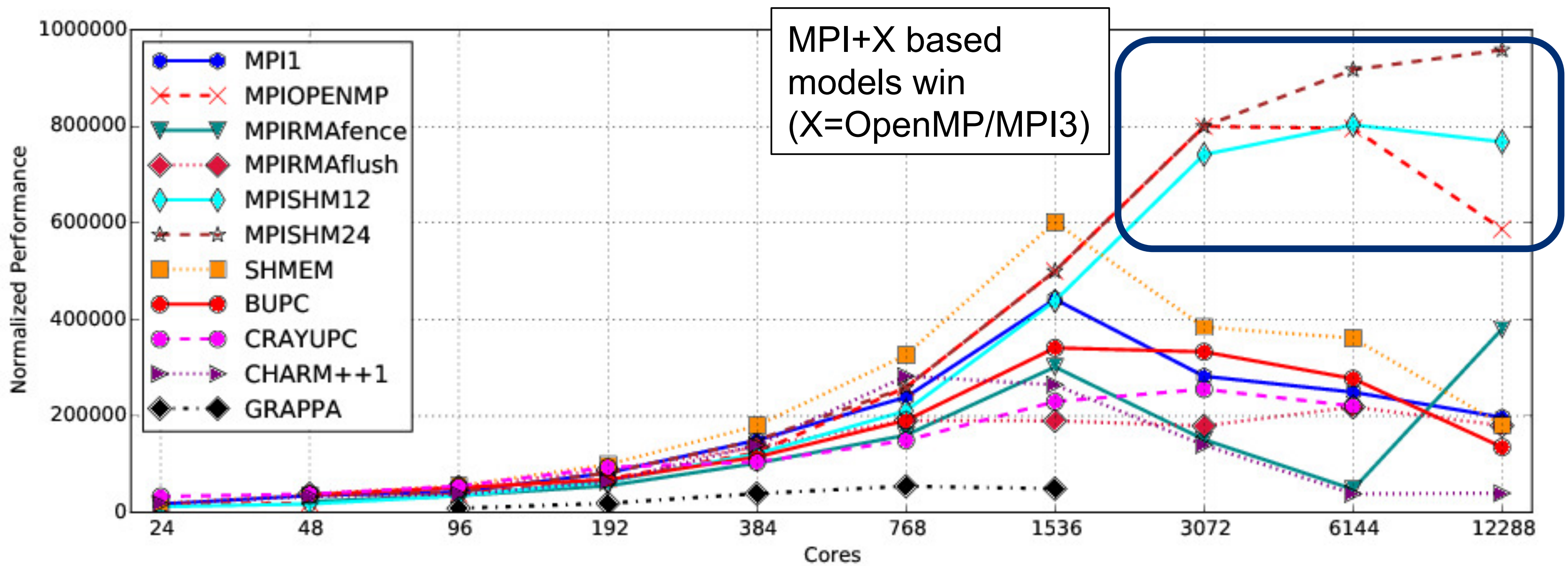
- Motivation
- Limitations
- Philosophy
- Context
- Usage model
- Reference implementations
- PRKs you should care about
- PRK you may care about
- Example results

Results

Following results obtained on NERSC Cray XC30 (Edison)

- two 12-core Intel® Xeon® E5-2695 processors per node
- Aries interconnect in a Dragonfly topology.
- Intel 15.0.1.133 C/C++ compiler for all codes, except Cray Compiler Environment (CCE) 8.4.0.219 for Cray UPC, and GCC 4.9.2 was used for Grappa. Berkeley UPC compiler 2.20.2 was used with the same Intel C/C++ compiler. System library versions Cray MPT (MPI and SHMEM) 7.2.1, uGNI 6.0, and DMAPP 7.0.1

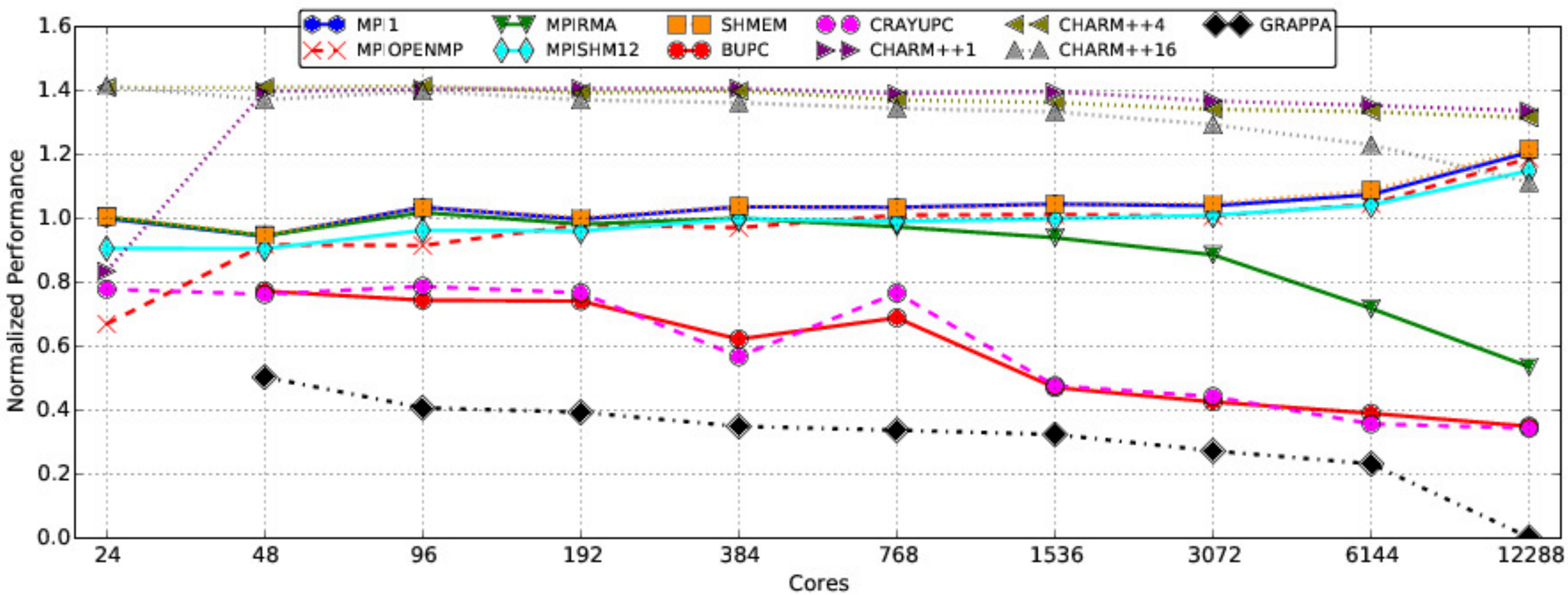
Transpose, strong scaled (49152x49152*)



Aggregate performance MB/s

* Charm++: (47104x47104)

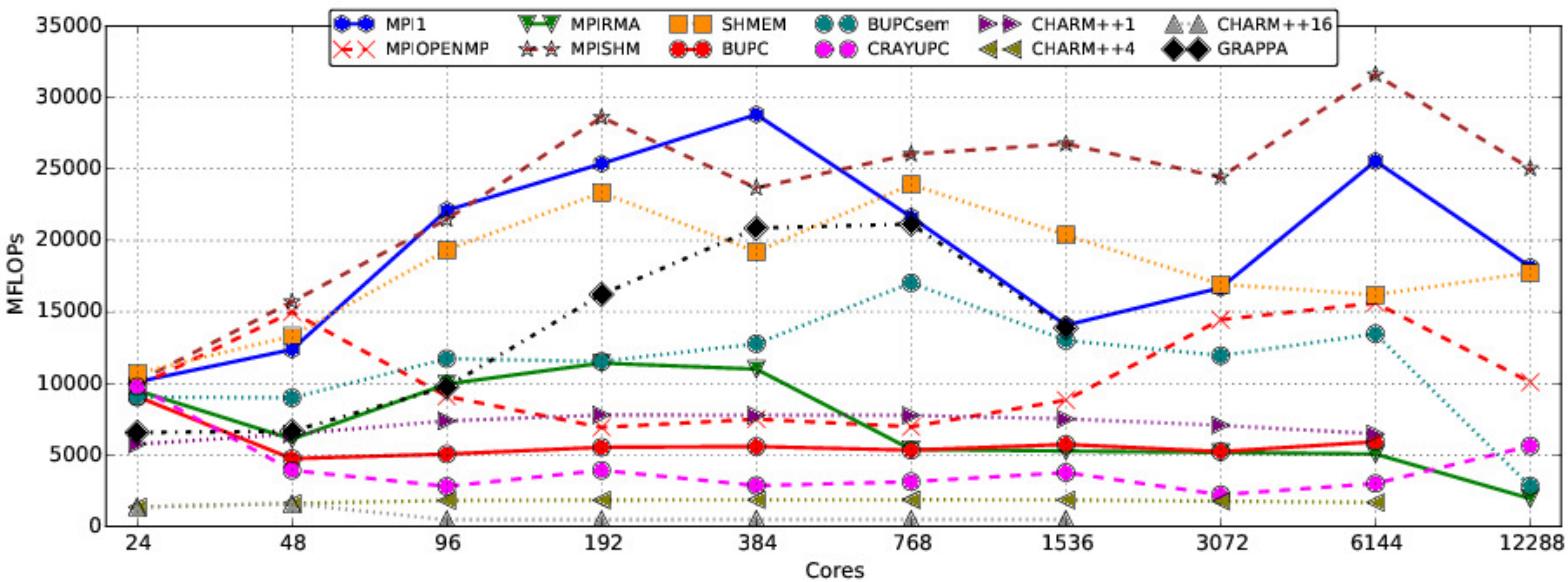
Stencil, radius=4, strong scaled (49152x49152*)



Normalized performance (Mflops/#nodes)/Mflops_single_node_MPI1

* Charm++: (47104x47104)

Synch_p2p, strong scaled (49152x49152*)



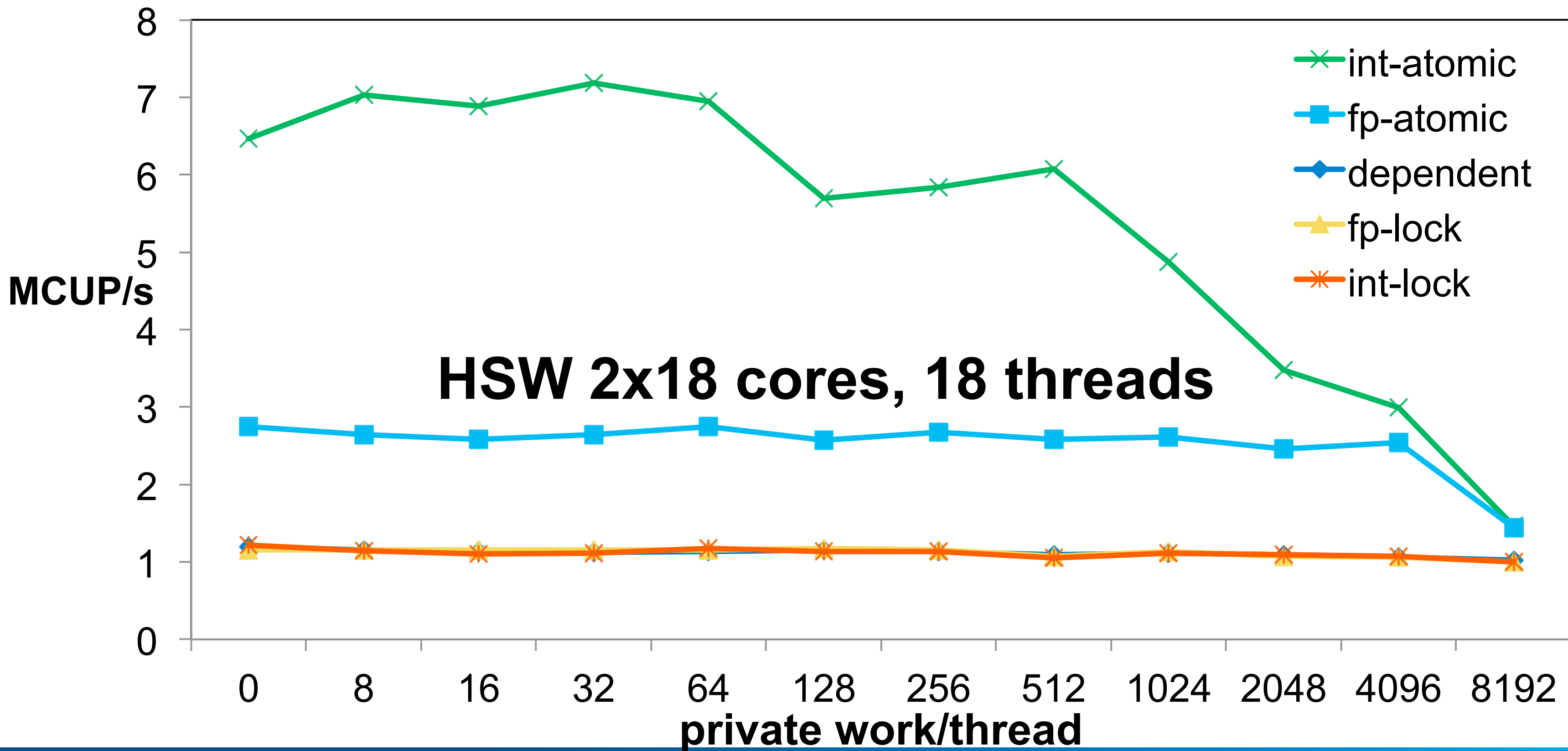
Aggregate performance MFlops

Results

Following results obtained on Xeon workstation

- two 18-core Intel® Xeon ® CPU E5-2699 processors per node
- Intel 17.0.0.098 C/C++ compiler with OpenMP enabled
- All 18 cores used on exactly one processor
- KMP_AFFINITY=granularity=fine,proclist=[{0,36},{1,37},{2,38},{3,39},{4,40},{5,41},{6,42},{7,43},{8,44},{9,45},{10,46},{11,47},{12,48},{13,49},{14,50},{15,51},{16,52},{17,53},{18,54},{19,55},{20,56},{21,57},{22,58},{23,59},{24,60},{25,61},{26,62},{27,63},{28,64},{29,65},{30,66},{31,67},{32,68},{33,69},{34,70},{35,71}],explicit (i.e. 1 thread/core)

Refcount results, shared counters



Summary

- PRK can be used to compare different aspects of parallel programming environments
- Growing set of reference implementations available: https://github.com/ParRes/Kernels
- Join the PRK community to contribute or review implementations!